

NintendoWare for Cafe

Sound Programming Guide

2015/07/03

Version 1.0.4

**The content of this document is highly confidential
and should be handled accordingly.**

Confidential

These coded instructions, statements, and computer programs contain proprietary information of Nintendo and/or its licensed developers and are protected by national and international copyright laws. They may not be disclosed to third parties or copied or duplicated in any form, in whole or in part, without the prior written consent of Nintendo.

Table of Contents

1	Introduction	7
1.1	Supported Platforms	7
1.2	Multi-Core and Multi-Thread Operations	7
2	The Sound Program Development Environment.....	8
2.1	The SoundLibrary	8
2.2	Directory Structure.....	8
2.3	Development Environment Used with the Sound Library.....	8
2.3.1	Configure the NintendoWare Build Environment	9
2.3.2	Configuring the Library File Links.....	9
2.3.3	Configuring the Include Path	9
2.3.4	Placing the Sound Data.....	9
2.3.5	Include the Sound ID File	10
3	Quick Start	11
3.1	Building and Running the Sample Program	11
3.2	Source Code.....	11
3.2.1	Referencing Header Files.....	17
3.2.2	Namespace	18
3.2.3	Platform-Specific Processes	18
3.3	The Initialization Process.....	19
3.3.1	Initialization of the Sound System	19
3.3.2	Initialization of the Sound Archive	20
3.3.3	Initialization of the Sound Data Manager	21
3.3.4	Initialization of the Sound Archive Player	21
3.3.5	Construction of the Sound Heap	22
3.4	Loading Sound Data.....	22
3.5	Frame Processing.....	23
3.6	Starting and Stopping Sound Playback.....	23
3.6.1	Sound Playback.....	23
3.6.2	Sound Handles	24
3.6.3	Stopping Sound Playback	24
3.7	Hold Play	25
3.7.1	Using the HoldSound Function.....	25
3.7.2	Resume Playback After Losing Player Priority.....	25
3.7.3	Player Priority Processing with the HoldSound Function	25
3.8	Prepare Sounds (Prepare for Playback)	26
3.8.1	Using the PrepareSound Function	26

3.9	Playback Using a String	26
3.10	Types of Sounds	27
4	Memory Management	29
4.1	How to Use the Sound Heap	29
4.1.1	Clear All	29
4.1.2	Restore Prior State	29
4.2	Sound Heap Application	30
4.2.1	Feature to Automatically Stop Invalid Sound	30
4.2.2	Multiple Sound Heaps	30
4.2.3	SoundMemoryAllocatable Class	30
4.3	Player Heap	31
4.3.1	The Player Heap	31
4.3.2	Using the Player Heap	31
4.3.3	Memory for the Player Heap	31
4.3.4	Cautions for Playback with Different Players	32
4.3.5	Data Loaded in the Player Heap	32
5	Sound Actors	34
5.1	Sound Actors	34
5.2	The SoundActor Program	34
5.2.1	Initializing the Instances	34
5.2.2	Sound Playback Using Sound Actors	34
5.2.3	Setting the Maximum Number of Sounds to Be Played Simultaneously	35
5.2.4	Parameter Updates	35
5.2.5	Deallocating Sound Actors	35
5.3	Actor Players	35
5.3.1	Maximum Number of Simultaneous Sounds Played and the Actor Player	35
5.3.2	Setting the Number of the Actor Player to Be Played	36
5.4	Dynamically Changing the Sound ID	36
5.4.1	The SoundActor::SetupSound Function	36
5.4.2	Changing the Sound ID	37
6	3D Sound	38
6.1	3D Sound Structure	38
6.1.1	The 3D Sound Manager (nw::snd::Sound3DManager)	38
6.1.2	The 3D Sound Actor (nw::snd::Sound3DActor)	38
6.1.3	The 3D Sound Listener (nw::snd::Sound3DListener)	38
6.2	3D Sound Program	38
6.2.1	Creating Instances	38
6.2.2	Initializing the 3D Sound Manager	39

6.2.3	Initializing the Listener	40
6.2.4	Initialize the Actor	42
6.2.5	3D Sound Playback	43
6.2.6	Updating the Actor's Coordinates	43
6.2.7	Lifespans of Actors and Sounds	43
6.3	Doppler Effect	43
6.3.1	Doppler Effect Parameters	44
6.3.2	Changing the Tone	44
6.3.3	Explicitly Specifying Speed	45
6.3.4	Formula for Pitch Change	45
6.4	Multi-Listeners	45
6.4.1	Differences Compared to a Single Listener	46
6.5	3D Sound Customization	46
6.5.1	3D Sound Engines (Sound3DEngine)	46
6.5.2	Creating a 3D Sound Engine	46
6.5.3	Implementing 3D Sound Calculation Processing	46
6.5.4	Sound3DCalculator	48
	Revision History	49

Code

Code 2-1	Library File Linking Configuration	9
Code 3-1	Building and Running the Sample Program	11
Code 3-2	Simple Demo Main.cpp	11
Code 3-3	Referenced Header Files	18
Code 3-4	Platform-Specific Macro	18
Code 3-5	Initializing the Sound System	19
Code 3-6	Initializing the Sound Archive	20
Code 3-7	Initializing the Sound Data Manager	21
Code 3-8	Initializing the Sound Archive Player	21
Code 3-9	Constructing the Sound Heap	22
Code 3-10	Loading Sound Data	22
Code 3-11	Frame Processing	23
Code 3-12	Sound Playback	23
Code 3-13	Stopping Sound Playback	24
Code 3-14	Hold Play of Sound	25
Code 3-15	Playback Using a Sound ID	26
Code 3-16	Playback Using a String	26
Code 3-17	Loading Label String Data	27
Code 3-18	Using Sequence Sound Handles	28
Code 5-1	SoundActor Instance Initialization	34
Code 5-2	Sound Playback Using Sound Actors	35

Code 5-3 Setting the Maximum Number of Sounds to Be Played Simultaneously	35
Code 5-4 Parameter Updates	35
Code 5-5 The SoundActor::SetupSound Function.....	36
Code 5-6 Dynamically Changing the Sound ID.....	37
Code 6-1 Creation of 3D Sound-Related Instances.....	38
Code 6-2 3D Sound Manager Initialization	39
Code 6-3 Initializing the Listener	40
Code 6-4 Calculating the Listener Matrix	40
Code 6-5 Initialize the Actor	42
Code 6-6 3D Sound Playback	43
Code 6-7 Updating Actor Coordinates	43
Code 6-8 Setting the Speed of Sound	44
Code 6-9 3D Sound Engine Inheritance	46
Code 6-10 3D Sound Engine Registration	46
Code 6-11 Overriding the UpdateAmbientParam Function.....	46
Code 6-12 The SoundAmbientParam Structure	47
Code 6-13 The Sound3DCalculator Class	48

Tables

Table 3-1 Platform-Specific Macros	18
--	----

Figures

Figure 2-1 Directory Structure	8
Figure 3-1 Sound and Sound Handle Are Tied Together When Sound Playback Succeeds.....	24
Figure 4-1 The LoadState Function Can Be Used to Restore a Prior State	30
Figure 5-1 Actor Player.....	36
Figure 6-1 Reducing the Player Priority Value	40
Figure 6-2 Maximum Volume Distance	41
Figure 6-3 Attenuation Unit Distance	42
Figure 6-4 Doppler Effect	44
Figure 6-5 Automatic Calculation of Speed.....	45

1 Introduction

The NintendoWare for Cafe (NintendoWare) sound library is for sound playback in Cafe applications. Use of the library enables sound data created with NWF4 SoundMaker (SoundMaker) and other tools to be played on the Cafe system.

This document explains the matters necessary for using the sound library to assemble programs that play sounds. It first describes how to build the development environment, and then walks through the sample program to explain by way of concrete examples how to use the library.

For more information about the individual classes and functions, see the Function Reference Manual.

1.1 Supported Platforms

The sound library targets the Cafe hardware and can run on the CAT-DEV. You can also use the library for development on a Windows PC. However, applications that run on a Windows PC might not run as is on the Cafe hardware.

This document describes the environment settings and build procedure for the CAT-DEV.

1.2 Multi-Core and Multi-Thread Operations

The sound library is not thread-safe. For this reason, the sound library API functions must be called from the same thread, and blocking must be implemented so that they are not called from another thread.

However, as indicated in the `subCore` demo, the internal processing of sounds can run on a separate core.

When this is the case, the initialization of the CAFE SDK AX library used internally by the sound library and the initialization of `nw::snd::SoundSystem` must take place on the same core.

2 The Sound Program Development Environment

This chapter explains how to build the sound program development environment.

2.1 The SoundLibrary

The sound library is used to assemble sound programs. The sound library corresponds to the functions of the `nw::snd` namespace included in the NintendoWare package.

2.2 Directory Structure

The directory structure of the sound library is shown below. In this figure, `%NW4F_ROOT%` denotes the directory `NW_Cafe` where NintendoWare was installed.

Figure 2-1 Directory Structure

```
%NW4F_ROOT%
|
+-- Library\Snd\           // Sound library source code and library files
+-- Demo\snd\             // Sample program
+-- DiscRoot\snd\         // Data for the sample program
+-- Document\Sound\       // Sound development environment documentation
|
+-- SampleData\Sound\     // Sample data for SoundMaker
+-- Tool\AnimSoundMaker\  // AnimSoundMaker tool
+-- Tool\SoundMaker\      // SoundMaker tool
+-- Tool\SoundPlayer\     // SoundPlayer tool
```

2.3 Development Environment Used with the Sound Library

The preparations required for development using the sound library can be broadly classified into these five steps:

1. Configure the build environment.
2. Configure the library file links.
3. Configure the include path.
4. Place the sound data.
5. Reference the sound ID file.

Below we present an example of the use the sound library using the NintendoWare build system. Specific references include `%NW4F_ROOT%\Demo\snd\simple\Makefile`.

2.3.1 Configure the NintendoWare Build Environment

First you must configure the NintendoWare build environment. See the Sound Quick Start document. If you need to read about this in greater detail, see the Runtime Library Startup Guide and the Build System documents.

2.3.2 Configuring the Library File Links

To use the sound library, you must link the sound library and anylibrary files that depend on the sound library. Use a linking configuration similar to the following in your makefile.

Code 2-1 Library File Linking Configuration

```
NW_USE_LIBS = SYS SND
```

The sound library depends on AX and other libraries of the `Cafe_SDK` and the NintendoWare system libraries.

If you are using the NintendoWare build system, you do not need to explicitly specify the Cafe SDK libraries. To link the NintendoWare system libraries, specify `SYS` for `NW_USE_LIBS`. To specify the sound library, specify `SND` for `NW_USE_LIBS`.

If you are not using NintendoWare build system, link `libnw4f_snd.a` (which is located below `%NW4F_ROOT%Library\Snd\lib\CAFE\{Debug, Develop, Release}`) as well as `libnw4f_sys.a` (which is located below `%NW4F_ROOT%\Library\Sys\lib\CAFE\{Debug, Develop, Release}`).

2.3.3 Configuring the Include Path

If you are using the NintendoWare build system, you do not need to set a special include path for the sound library.

If you are not using NintendoWare build system, please add `%NW4F_ROOT%Library\Snd\include` and `%NW4F_ROOT%Library\Sys\include` to the include path.

2.3.4 Placing the Sound Data

The sound data created by the sound designer is passed to the programmer as a folder containing the Sound Archive (`*.bfsar`) and multiple sets of stream data (`*.bfstm`). The Sound Archive is a single file that groups multiple sets of sound data other than stream data.

The programmer places this sound data in storage to make use of it. The programmer just needs to place the necessary files below the directory specified in the environment variable `%CAFE_CONTENT_DIR%`. The sound data for the simple demo (described later) is located in `%NW4F_ROOT%\DiscRoot\snd\common`.

2.3.5 Include the Sound ID File

The Sound ID file (`*.fsid`) contains definitions of labels for using the Sound Archive. This file is exported at the same time that the Sound Archive is created. Like the header files, the programmer includes this by referencing it in the source file.

In the simple demo, this corresponds to `%NW4F_ROOT%\Demo\snd\simple\common.fsid`.

3 Quick Start

This chapter uses the sample program by way of example to explain how to build a simple sound program.

First, you will build and execute the sample demo, and then you will look at the sample's source code while reviewing how to assemble a sound program.

3.1 Building and Running the Sample Program

This section provides a very simple explanation of how to build and run the demo %NW4F_ROOT%\Demo\snd\simple.

To read details about building, running and debugging, see the Sound Sample Program document.

Code 3-1 Building and Running the Sample Program

```
$ cd %NW4F_ROOT%\Demo\snd\simple      # Move to sample program directory
$ make                                # Build
$ make run                             # Run
```

If the sample program runs successfully, you can play sounds using the A Button, X Button and other operations of the Classic Controller Pro connected to the CAT-DEV hardware.

3.2 Source Code

This section explains the sound program by showing the contents of the source code.

The code shown below is from the main.cpp file located in the %NW4F_ROOT%\Demo\snd\simple directory.

Code 3-2 Simple Demo Main.cpp

```
#include <nw/snd.h>
#include <nw/demo.h>
#include "common.fsid"

#ifdef NW_PLATFORM_CAFE
#include <cafe.h>
#else
#include <winext/cafe.h>
#include <Windows.h>
#endif

// #define CPU_RENDERING

namespace
{
    const char CONTENT_DIR[] = "/vol/content";
    const char SOUND_ARC_PATH[] = "snd/common/common.bfsar";
```

```

const s32 SOUND_HEAP_SIZE = 4 * 1024 * 1024;

nw::snd::FsSoundArchive      s_SoundArchive;
nw::snd::SoundArchivePlayer  s_SoundArchivePlayer;
nw::snd::SoundDataManager    s_SoundDataManager;
nw::snd::SoundHeap           s_SoundHeap;
nw::snd::SoundHandle          s_SoundHandle;
nw::snd::SoundHandle          s_SoundHandleHold;

void* s_pMemoryForSoundSystem;
void* s_pMemoryForInfoBlock;
void* s_pMemoryForSoundDataManager;
void* s_pMemoryForSoundArchivePlayer;
void* s_pMemoryForSoundHeap;
void* s_pMemoryForStreamBuffer;

#if defined(NW_PLATFORM_CAFE)
FSClient* s_pFsClient;
void* s_pMemoryForFsSoundArchive;
#endif

void InitializeNwSound()
{
    // Initialize sound system
    nw::snd::SoundSystem::SoundSystemParam param;
    size_t workMemSize =
nw::snd::SoundSystem::GetRequiredMemSize( param );
    nw::demo::DefaultAllocator allocator;
    s_pMemoryForSoundSystem = allocator.Alloc( workMemSize, 4 );

    nw::snd::SoundSystem::Initialize(
        param,
        reinterpret_cast<uptr>( s_pMemoryForSoundSystem ),
        workMemSize );

    // Initialize sound archive
    {
        #if defined(NW_PLATFORM_CAFE)
            size_t size = s_SoundArchive.GetRequiredMemSize();
            s_pMemoryForFsSoundArchive = allocator.Alloc(size);
            if ( ! s_SoundArchive.Open(
                s_pFsClient, SOUND_ARC_PATH, s_pMemoryForFsSoundArchive,
size) )
            {
                NW_ASSERTMSG( 0, "cannot open bfsar(%s)\n", SOUND_ARC_PATH );
            }
        #else
            if ( ! s_SoundArchive.Open(SOUND_ARC_PATH) )
            {
                NW_ASSERTMSG( 0, "cannot open bfsar(%s)\n", SOUND_ARC_PATH );
            }
        #endif
    }
}

```

```

#endif
}

// Load INFO block
{
    size_t infoBlockSize = s_SoundArchive.GetHeaderSize();
    s_pMemoryForInfoBlock = allocator.Alloc(
        infoBlockSize, nw::snd::FsSoundArchive::BUFFER_ALIGN_SIZE );
    if ( ! s_SoundArchive.LoadHeader(
        s_pMemoryForInfoBlock, infoBlockSize ) )
    {
        NW_ASSERTMSG( 0, "cannot load infoBlock(%s)",
SOUND_ARC_PATH );
    }
}

// Initialize sound data manager
{
    size_t setupSize =
        s_SoundDataManager.GetRequiredMemSize( &s_SoundArchive );
    s_pMemoryForSoundDataManager = allocator.Alloc( setupSize, 4 );
    s_SoundDataManager.Initialize(
        &s_SoundArchive, s_pMemoryForSoundDataManager, setupSize );
}

// Initialize sound archive player
{
    size_t setupSize =
        s_SoundArchivePlayer.GetRequiredMemSize( &s_SoundArchive );
    s_pMemoryForSoundArchivePlayer = allocator.Alloc( setupSize, 32 );
    size_t setupStrmBufferSize =
s_SoundArchivePlayer.GetRequiredStreamBufferSize(&s_SoundArchive);
    s_pMemoryForStreamBuffer = allocator.Alloc( setupStrmBufferSize,
8 );
    bool result = s_SoundArchivePlayer.Initialize(
        &s_SoundArchive,
        &s_SoundDataManager,
        s_pMemoryForSoundArchivePlayer, setupSize,
        s_pMemoryForStreamBuffer, setupStrmBufferSize );
    NW_ASSERT( result );
}

// Constuct sound heap
{
    s_pMemoryForSoundHeap = allocator.Alloc( SOUND_HEAP_SIZE );
    bool result = s_SoundHeap.Create(
        s_pMemoryForSoundHeap, SOUND_HEAP_SIZE );
    NW_ASSERT( result );
}

```

```

    // Load data
    if ( ! s_SoundDataManager.LoadData( SEQ_MARIOKART, &s_SoundHeap ) )
    {
        NW_ASSERTMSG( false, "LoadData(SEQ_MARIOKART) failed." );
    }
    if ( ! s_SoundDataManager.LoadData( SE_YOSHI, &s_SoundHeap ) )
    {
        NW_ASSERTMSG( false, "LoadData(SE_YOSHI) failed." );
    }
}

} // namespace

int
NwDemoMain(int argc, char **argv)
{
#ifdef NW_PLATFORM_CAFE
    u32 gx2InitAttribs[] =
    {
        GX2_INIT_ATTRIB_ARGC,      (u32)argc,    // number of args
        GX2_INIT_ATTRIB_ARGV,      (u32)argv,    // command-line args
        GX2_INIT_ATTRIB_NULL,      // terminates the list
    };
    GX2Init(gx2InitAttribs);

    AXInit();

#ifdef CPU_RENDERING
    AXSetDefaultMixerSelect( AX_PB_MIXER_SELECT_PPC );
#else
    AXSetDefaultMixerSelect( AX_PB_MIXER_SELECT_DSP );
#endif
#endif

    nw::demo::DefaultAllocator allocator;
    nw::demo::DemoSystem::CreateArg demoSystemArg;
    demoSystemArg.allocator = &allocator;
#ifdef NW_PLATFORM_WIN32
    demoSystemArg.waitVBlank = 0;
#endif
    nw::demo::DemoSystem* pDemo =
        new( allocator.Alloc( sizeof( nw::demo::DemoSystem ) ) )
        nw::demo::DemoSystem( demoSystemArg );
    pDemo->Initialize();
    pDemo->InitializeGraphicsSystem();

#ifdef NW_PLATFORM_CAFE
    // Initialize FS
    {
        FSInit();
        s_pFsClient =

```

```

reinterpret_cast<FSClient*>(allocator.Alloc(sizeof(FSClient)));
    std::memset(s_pFsClient, 0x00, sizeof(FSClient));
    NW_ASSERT_NOT_NULL(s_pFsClient);

    FSAddClient(s_pFsClient, FS_RET_NO_ERROR);
}

// Move directory
{
    FSCmdBlock* block =

reinterpret_cast<FSCmdBlock*>(allocator.Alloc(sizeof(FSCmdBlock)));
    NW_ASSERT_NOT_NULL(block);
    FSInitCmdBlock(block);

    FSStatus status =
        FSChangeDir(s_pFsClient, block, CONTENT_DIR, FS_RET_NO_ERROR);
    NW_ASSERT(status == FS_STATUS_OK);

    allocator.Free(block);
}
#else
// Move directory
{
    static const int cFileNameLen = 512;
    char buf[cFileNameLen];
    if ( GetEnvironmentVariableA("NW4F_ROOT", buf, cFileNameLen) > 0 )
    {
        nw::ut::snprintf(buf, cFileNameLen, "%s/DiscRoot", buf );
        NW_LOG("buf(%s)\n", buf);
        SetCurrentDirectoryA(buf);
    }
    else
    {
        NW_LOG("cannot read NW4F_ROOT\n");
        return -1;
    }
}
#endif

InitializeNwSound();

NW_LOG("-----\n");
NW_LOG("NW4F Demo/snd/simple\n");
NW_LOG("-----\n");
NW_LOG("[A]      StartSound SEQ  (SEQ_MARIOKART)\n");
NW_LOG("[X]      StartSound WSD   (SE_YOSHI)\n");
NW_LOG("[Y]      StartSound STRM  (STRM_MARIOKART)\n");
NW_LOG("[B]      Stop  Sound\n");
NW_LOG("[RIGHT] HoldSound SEQ  (SEQ_MARIOKART)\n");
NW_LOG("[UP]     HoldSound WSD   (SE_YOSHI)\n");

```

```

    NW_LOG("[LEFT] HoldSound STRM (STRM_MARIOKART)\n");
#if defined( NW_PLATFORM_CAFE )
    NW_LOG("[HOME] Exit Application\n");
#elif defined( NW_PLATFORM_WIN32 )
    NW_LOG("[S] Exit Application\n");
#endif
    NW_LOG("-----\n");

    while( true ) {
#if defined( NW_PLATFORM_CAFE )
        GX2WaitForVsync();
#else
        pDemo->WaitForVBlank();
#endif

        pDemo->UpdatePad();
        nw::demo::Pad* pad = pDemo->GetPad();

        // StartSound / Stop
        if ( pad->IsTrig( nw::demo::Pad::MASK_A ) ) {
            s_SoundHandle.Stop( 0 );
            bool result = s_SoundArchivePlayer
                .StartSound( &s_SoundHandle, SEQ_MARIOKART ).IsSuccess();
            NW_LOG("[SEQ] StartSound(SEQ_MARIOKART) ... (%d)\n", result);
        }
        if ( pad->IsTrig( nw::demo::Pad::MASK_X ) ) {
            s_SoundHandle.Stop( 0 );
            bool result = s_SoundArchivePlayer
                .StartSound( &s_SoundHandle, SE_YOSHI ).IsSuccess();
            NW_LOG("[WSD] StartSound(SE_YOSHI) ... (%d)\n", result);
        }
        if ( pad->IsTrig( nw::demo::Pad::MASK_Y ) ) {
            s_SoundHandle.Stop( 0 );
            bool result = s_SoundArchivePlayer
                .StartSound( &s_SoundHandle, STRM_MARIOKART ).IsSuccess();
            NW_LOG("[STRM] StartSound(STRM_MARIOKART) ... (%d)\n", result);
        }
        if ( pad->IsTrig( nw::demo::Pad::MASK_B ) ) {
            s_SoundHandle.Stop( 0 );
        }

        // HoldSound
        if ( pad->IsHold( nw::demo::Pad::MASK_RIGHT ) ) {
            bool result = s_SoundArchivePlayer
                .HoldSound( &s_SoundHandleHold,
SEQ_MARIOKART ).IsSuccess();
            NW_LOG("[SEQ] HoldSound(SEQ_MARIOKART) ... (%d)\n", result);
        }
        if ( pad->IsHold( nw::demo::Pad::MASK_UP ) ) {
            bool result = s_SoundArchivePlayer
                .HoldSound( &s_SoundHandleHold, SE_YOSHI ).IsSuccess();
            NW_LOG("[WSD] HoldSound(SE_YOSHI) ... (%d)\n", result);
        }
    }

```



```

    }
    if ( pad->IsHold( nw::demo::Pad::MASK_LEFT ) ) {
        bool result = s_SoundArchivePlayer
            .HoldSound( &s_SoundHandleHold,
STRM_MARIOKART ).IsSuccess();
        NW_LOG("[STRM] HoldSound(STRM_MARIOKART) ... (%d)\n", result);
    }

    // Exit
    if ( pad->IsTrig( nw::demo::Pad::MASK_START ) ) {
        break;
    }

    s_SoundArchivePlayer.Update();
}

s_SoundHeap.Destroy();
s_SoundArchivePlayer.Finalize();
s_SoundDataManager.Finalize();
s_SoundArchive.Close();
nw::snd::SoundSystem::Finalize();

allocator.Free( s_pMemoryForSoundHeap );
allocator.Free( s_pMemoryForSoundArchivePlayer );
allocator.Free( s_pMemoryForStreamBuffer );
allocator.Free( s_pMemoryForSoundDataManager );
allocator.Free( s_pMemoryForInfoBlock );
allocator.Free( s_pMemoryForSoundSystem );

#ifdef NW_PLATFORM_CAFE
FSDelClient(s_pFsClient, FS_RET_NO_ERROR);
allocator.Free(s_pFsClient);
s_pFsClient = NULL;
FSShutdown();

AXQuit();

GX2Shutdown();
#endif

pDemo->FinalizeGraphicsSystem();
pDemo->Finalize();
allocator.Free( pDemo );

return 0;
}

```

3.2.1 Referencing Header Files

The `main.cpp` file references the following header files.

Code 3-3 Referenced Header Files

```
#include <nw/snd.h>
#include <nw/demo.h>
#include "common.fsid"

#if defined( NW_PLATFORM_CAFE )
#include <cafe.h>
#else
#include <winext/cafe.h>
#include <Windows.h>
#endif
```

The `nw/snd.h` file is a header file for the sound library. This header file must be referenced in order to use the sound library. The `common.fsid` file is a Sound ID file. By using the labels defined in this file, you can use specific sets of sound data in the Sound Archive.

The `nw/demo/demo_Main.h` file is included so `NwDemoMain` can be used as the entry point and for vertical synchronization with the PC version. If you are only running the program on the actual hardware, you do not need `NwDemoMain` or this header file, and it is all right to use the `main` function.

3.2.2 Namespace

The sound library is defined by the namespace `nw::snd`.

3.2.3 Platform-Specific Processes

Here and there you will see processes written as macros that look like this:

Code 3-4 Platform-Specific Macro

```
#if defined( NW_PLATFORM_CAFE )
:
#else
:
#endif
```

By including the `nw/types.h` file, macros like those shown below are defined to indicate the platform (CAT-DEV or PC). By using these macros you can write platform-specific processes.

Table 3-1 Platform-Specific Macros

Macro	Description
<code>NW_PLATFORM_CAFE</code>	Defined when building with MULTI for the CAT-DEV.
<code>NW_PLATFORM_WIN32</code>	Defined when building with Visual C++ for the PC.

The sound library functions themselves do not differ for the CAT-DEV and the PC, but these macros are utilized to deal with differences at the SDK level.

In programs like the simple demo, the `nw/types.h` file is included indirectly when the `nw/demo.h` file is included.

3.3 The Initialization Process

The basic initialization process involves the following steps.

1. Initialization of the Sound System
2. Initialization of the Sound Archive
3. Initialization of the Sound Data Manager
4. Initialization of the Sound Archive Player
5. Construction of the Sound Heap

Each of these processes is explained in order.

3.3.1 Initialization of the Sound System

To initialize the Sound System, call the `nw::snd::SoundSystem` class of functions, as shown below:

Code 3-5 Initializing the Sound System

```
// Initialize sound system
nw::snd::SoundSystem::SoundSystemParam param;
size_t workMemSize = nw::snd::SoundSystem::GetRequiredMemSize( param );
nw::demo::DefaultAllocator allocator;
s_pMemoryForSoundSystem = allocator.Alloc( workMemSize, 4 );

nw::snd::SoundSystem::Initialize(
    param,
    reinterpret_cast<uptr>( s_pMemoryForSoundSystem ),
    workMemSize );
```

The `nw::snd::SoundSystem::Initialize` function initializes the Sound System, and doing this starts the sound thread and the sound data load thread which run inside the sound library.

For the arguments, pass the priority of the sound thread and the sound data load thread and the `nw::snd::SoundSystem::SoundSystemParam` structure, which holds such information as the stack sizes of the threads. The sound thread handles sound playback. To prevent delays in playback, the sound thread must be set to a high priority.

The sound data load thread handles the loading of stream data and the loading of data to the player heap. If there are delays in the loading of stream data there will be interruptions in sound, so the sound data load thread must also be set to a high priority. To prevent sound breaks, we recommend setting the sound data load thread to a higher priority than the sound thread.

`nw::demo::DefaultAllocator` is used here to allocate memory with the PC version. You can also use `MemAllocFromDefaultHeap` when you are running the program with only the actual hardware.

3.3.2 Initialization of the Sound Archive

Next the Sound Archive is initialized, as shown below.

Code 3-6 Initializing the Sound Archive

```
nw::snd::FsSoundArchive    s_SoundArchive;

#if defined(NW_PLATFORM_CAFE)
FSClient* s_pFsClient;
void* s_pMemoryForFsSoundArchive;
#endif

// Initialize sound archive
{
#if defined(NW_PLATFORM_CAFE)
    size_t size = s_SoundArchive.GetRequiredMemSize();
    s_pMemoryForFsSoundArchive = allocator.Alloc(size);
    if ( ! s_SoundArchive.Open(
        s_pFsClient, SOUND_ARC_PATH, s_pMemoryForFsSoundArchive, size) )
    { NW_ASSERTMSG( 0, "cannot open bfsar(%s)\n", SOUND_ARC_PATH ); }
#else
    if ( ! s_SoundArchive.Open(SOUND_ARC_PATH) )
    { NW_ASSERTMSG( 0, "cannot open bfsar(%s)\n", SOUND_ARC_PATH ); }
#endif
}

// Load INFO block
{
    size_t infoBlockSize = s_SoundArchive.GetHeaderSize();
    s_pMemoryForInfoBlock = allocator.Alloc(
        infoBlockSize, nw::snd::FsSoundArchive::BUFFER_ALIGN_SIZE );
    if ( ! s_SoundArchive.LoadHeader( s_pMemoryForInfoBlock,
infoBlockSize ) )
    {
        NW_ASSERTMSG( 0, "cannot load infoBlock(%s)", SOUND_ARC_PATH );
    }
}
```

The `nw::snd::FsSoundArchive` class instance `s_SoundArchive` is secured ahead of time. Open the Sound Archive using the `nw::snd::FsSoundArchive::Open` function. The argument specifies the path to the FS client and the FS filesystem, and the memory region required for the Sound Archive. To get the memory size required for the Sound Archive, use the `nw::snd::FsSoundArchive::GetRequiredMemSize` function.

Next, call the `nw::snd::FsSoundArchive::LoadHeader` function to load the minimum essential amount of information. The size of memory required for loading this information is passed as one of the function's arguments. Use the `nw::snd::FsSoundArchive::GetHeaderSize` function to get the required size of this region.

3.3.3 Initialization of the Sound Data Manager

The next step is to initialize the Sound Data Manager. This is the class for loading and managing the data stored in a sound archive.

Code 3-7 Initializing the Sound Data Manager

```
nw::snd::SoundDataManager  s_SoundDataManager;

// Initialize the sound data manager
{
    size_t setupSize =
        s_SoundDataManager.GetRequiredMemSize( &s_SoundArchive );
    s_pMemoryForSoundDataManager = allocator.Alloc( setupSize, 4 );
    s_SoundDataManager.Initialize(
        &s_SoundArchive, s_pMemoryForSoundDataManager, setupSize );
}
```

The `nw::snd::SoundDataManager` class instance `s_SoundDataManager` is secured ahead of time. Use the `nw::snd::SoundDataManager::Initialize` function to initialize the Sound Data Manager. The function arguments take `s_SoundArchive`, a pointer to the sound archive, and the size of the memory required to initialize the Sound Data Manager. Use the `nw::snd::SoundDataManager::GetRequiredMemSize` function to get the required size of this region.

3.3.4 Initialization of the Sound Archive Player

The next step is to initialize the Sound Archive Player. This is the class for using the Sound Archive to play sounds.

Code 3-8 Initializing the Sound Archive Player

```
nw::snd::SoundArchivePlayer s_SoundArchivePlayer;

// Initialize Sound Archive Player
{
    size_t setupSize =
        s_SoundArchivePlayer.GetRequiredMemSize( &s_SoundArchive );
    s_pMemoryForSoundArchivePlayer = allocator.Alloc( setupSize, 32 );
    size_t setupStrmBufferSize =

        s_SoundArchivePlayer.GetRequiredStreamBufferSize( &s_SoundArchive );
    s_pMemoryForStreamBuffer = allocator.Alloc( setupStrmBufferSize, 32 );
    bool result = s_SoundArchivePlayer.Initialize(
        &s_SoundArchive,
        &s_SoundDataManager,
        s_pMemoryForSoundArchivePlayer, setupSize,
        s_pMemoryForStreamBuffer, setupStrmBufferSize );
    NW_ASSERT( result );
}
```

The `nw::snd::SoundArchivePlayer` class instance `s_SoundArchivePlayer` is secured ahead of time.

The Sound Archive Player is initialized by the `nw::snd::SoundArchivePlayer::Initialize` function. The function arguments are: the `s_SoundArchive` pointer to the Sound Archive, the `s_SoundDataManager` pointer to the Sound Data Manager, and two memory regions required for initialization. One of these memory regions is used as a work area and for instances sounds kept in the sound archive player. The other memory region is used as a buffer for playback of stream sounds.

The sizes of these memory regions can be obtained using `nw::snd::SoundArchivePlayer::GetRequiredMemSize` and `nw::snd::SoundArchivePlayer::GetRequiredStreamBufferSize`.

3.3.5 Construction of the Sound Heap

The final process is to construct the sound heap. This is the class for managing the memory regions for loading sound data.

Code 3-9 Constructing the Sound Heap

```
nw::snd::SoundHeap      s_SoundHeap;

// Construct the sound heap
{
    s_pMemoryForSoundHeap = allocator.Alloc( SOUND_HEAP_SIZE );
    bool result = s_SoundHeap.Create( s_pMemoryForSoundHeap, SOUND_HEAP_SIZE );
    NW_ASSERT( result );
}
```

The `nw::snd::SoundHeap` class instance `s_SoundHeap` is secured ahead of time.

The sound heap is built by the `nw::snd::SoundHeap::Create` function. The function takes the memory region assigned to the sound heap as an argument.

3.4 Loading Sound Data

To play wave sounds and sequence sounds, you need to load the sound data needed for playback ahead of time. The sound data can be loaded in set units of groups, or it can be loaded in units of sequence data, wave sound data, bank data, and waveform archive data. The determination of what data to include in a group is configured on SoundMaker.

Code 3-10 Loading Sound Data

```
// Load data
if ( ! s_SoundDataManager.LoadData( SEQ_MARIOKART, &s_SoundHeap ) )
{
    NW_ASSERTMSG( false, "LoadData(SEQ_MARIOKART) failed." );
}
if ( ! s_SoundDataManager.LoadData( SE_YOSHI, &s_SoundHeap ) )
{
```

```
NW_ASSERTMSG( false, "LoadData(SE_YOSHI) failed." );  
}
```

You can use the `nw::snd::SoundDataManager::LoadData` function to load the sound data. The arguments take the sound data to load or the group's label, and the sound heap for storing the loaded data.

The group labels `SEQ_MARIOKART` and `SE_YOSHI` are defined by the Sound ID file `common.fsid`. Confirm with the sound designer which data should be loaded at which times.

The simple demo loads all data for the sequence sound `SEQ_MARIOKART` (sequence data, bank data, waveform archive data) and all data for the wave sound `SE_YOSHI` (wave sound data, waveform archive data).

Note: The `nw::snd::SoundDataManager::LoadData` function performs synchronous loading. No asynchronous version of this function is provided. To perform asynchronous loading, call this function from another thread. Be sure to refer to the reference manual for precautions when calling the `nw::snd::SoundDataManager::LoadData` function from another thread.

3.5 Frame Processing

The Sound Archive Player must be updated in every frame.

Code 3-11 Frame Processing

```
s_SoundArchivePlayer.Update();
```

This update process is normally called from the main loop. It does not need to be called in every video frame.

For processes like fading the volume, a call to this function is figured as 1 frame.

3.6 Starting and Stopping Sound Playback

3.6.1 Sound Playback

Sounds are played using code like the following.

Code 3-12 Sound Playback

```
bool result = s_SoundArchivePlayer  
    .StartSound( &s_SoundHandle, SEQ_MARIOKART ).IsSuccess();
```

The `nw::snd::SoundArchivePlayer::StartSound` function plays sounds. The arguments take the sound handle and the label for the sound being played.

The `nw::snd::SoundHandle` class instance pointer `s_SoundHandle` is passed as the sound handle (explained in section 3.6.2 Sound Handles).

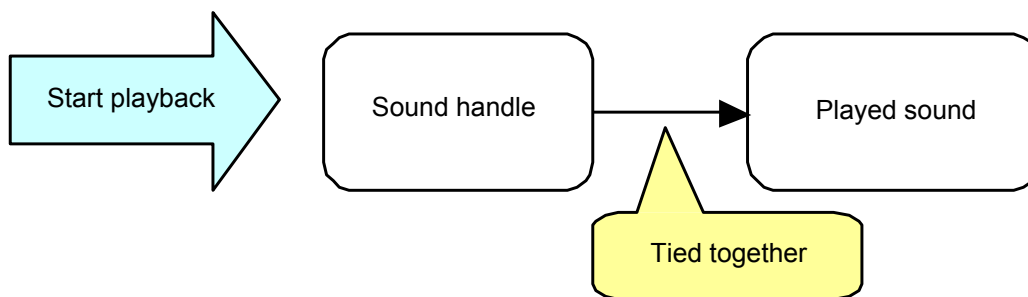
The label for the sound being played is defined in the Sound ID file common.fsid. You need to confirm with the sound designer which label is for which sound.

3.6.2 Sound Handles

3.6.2.1 What Is a Sound Handle?

A sound handle is an object for stopping sounds that are playing, changing the volume, and other functions. Each sound handle can control one sound. When the sound starts playing successfully, the sound and a sound handle become tied together. Until that tie is cut, the sound handle can be used to control that sound.

Figure 3-1 Sound and Sound Handle Are Tied Together When Sound Playback Succeeds



This means that the programmer does not need to be concerned with whether a sound is still playing or not. You can carry out the same process on a sound whether it is playing or already stopped without mistakenly performing the operation on a different sound.

3.6.2.2 Tips for Creation of Sound Handles

For sounds like one-time sound effects that play without stopping or being manipulated, you only need to prepare one sound handle, using it repeatedly to play each sound in turn. Also, you can use it immediately after a sound to change the volume and other parameters before playing the next sound.

For persistent effects like background music and engine noise, you'll at least need to stop the sound at some point, so you will need a sound handle for each individual sound.

3.6.3 Stopping Sound Playback

A sound handle is used to stop its associated sound.

Code 3-13 Stopping Sound Playback

```
s_SoundHandle.Stop( 0 );
```

Use the `nw::snd::SoundHandle::Stop` function to stop sounds. The argument takes the number of fadeout frames. The volume is gradually decreased for the duration of the specified number of fadeout frames and then playback is stopped. The number of fadeout frames corresponds to the number of calls to the `nw::snd::SoundArchivePlayer::Update` function.

As mentioned above, it is okay if the sound stops before the function is called. If this occurs, the function returns without doing anything.

3.7 Hold Play

Normally, the `StartSound` and `Stop` functions are used to play back and stop sound. However, it is sometimes troublesome to call the `Stop` function at the appropriate time when implementing the program. In these cases, the `HoldSound` function can be used in place of the `StartSound` function.

3.7.1 Using the HoldSound Function

When using the `HoldSound` function, it must be called for each frame while the sound is playing. As this happens, the playback start process is performed the first time only, and no processing is done the second or subsequent times. This method of playback is called Hold Play in the NW4F sound library.

Sound continues to play while the `HoldSound` function is being called, but as soon as the function stops being called, the sound stops automatically inside the

`nw::snd::SoundArchivePlayer::Update` function.

Code 3-14 Hold Play of Sound

```
bool result = s_SoundArchivePlayer
    .HoldSound( &s_SoundHandleHold, SEQ_MARIOKART ).IsSuccess();
```

With this notation, the `HoldSound` function is only called in place of the `StartSound` function.

3.7.2 Resume Playback After Losing Player Priority

When the `StartSound` function and `Stop` function are used for playback, if sound playback is forcibly stopped due to loss of player priority, playback will not resume unless `StartSound` is explicitly called.

When using `HoldSound`, even if player priority is lost, an attempt to resume playback will occur in the next frame because the function is called in each frame. For this reason, playback automatically resumes when the playback of a higher priority sound completes.

However, because playback resumes from the beginning rather than from the middle, be aware that some sound data may sound unnatural.

3.7.3 Player Priority Processing with the HoldSound Function

When the `HoldSound` function is used, sound playback player priority processing differs from normal processing. Assuming that the player priority value set is 64, when playback starts the priority is 63 (one less than 64). If playback is successful, then the player priority is set to the value of 64.

Normally the latter of two processes with the same player priority takes precedence, but for this reason, when `HoldSound` is used the former takes precedence.

3.8 Prepare Sounds (Prepare for Playback)

Depending on the sound data type, playback may not start immediately even if the `StartSound` function is called. This is the case when playing a stream sounds: Because some data must be pre-loaded, playback does not begin until loading completes. In addition, when you use the player heap to play wave sounds and sequence sounds, playback will not begin until the necessary data are finished loading into the player heap. (To read about the player heap, see 4.3 Player Heap.) Normally, this is not a major issue, but there are cases when a problem can arise if the timing of the start of playback is inconsistent, such as when synchronizing images and sound. In these cases, the `PrepareSound` function can be used in place of the `StartSound` function.

3.8.1 Using the PrepareSound Function

The `PrepareSound` function performs only the preparation for starting sound playback. For this reason, sound will not be played by calling only the `PrepareSound` function.

Preparation for starting sound playback occurs asynchronously. Use the `nw::snd::SoundHandle::IsPrepared` function to verify whether preparations have completed. After confirming that preparations are finished, start sound playback by calling the `nw::snd::SoundHandle::StartPrepared` function.

If preparations are complete, call the `StartPrepared` function to start sound playback immediately. If the `StartPrepared` function is called before preparations are complete, sound playback will begin automatically after waiting for preparations to complete.

3.9 Playback Using a String

In the sample code described previously, the sound ID of the sound to be played by the playback function was passed in the following ways.

Code 3-15 Playback Using a Sound ID

```
bool result = s_SoundArchivePlayer
    .StartSound( &s_SoundHandle, SEQ_MARIOKART ).IsSuccess();
```

You must include a sound ID file (*.fsid) ahead of time because this sound ID is defined in a sound ID file. This is a drawback because you need to recompile each time the sound ID file is updated. In addition to the above, a method of playing back sounds using a string is also provided.

Code 3-16 Playback Using a String

```
bool result = s_SoundArchivePlayer
    .StartSound( &s_SoundHandle, "STRM_MARIOKART" ).IsSuccess();
```

A sound ID file does not need to be included in this case because a string is used under this method. Note, however, that it is necessary to add the following type of initialization.

Code 3-17 Loading Label String Data

```
// Load STRING block
{
    u32 stringBlockSize = m_Archive.GetLabelStringDataSize();
    m_pMemoryForStringBlock = MemAlloc(
        stringBlockSize, nw::snd::FsSoundArchive::BUFFER_ALIGN_SIZE );
    if ( ! m_Archive.LoadLabelStringData(
        m_pMemoryForStringBlock, stringBlockSize ) )
    {
        NW_ASSERTMSG( 0, "cannot load stringBlock(%s)",
SOUND_ARC_PATH );
    }
}
```

Playback using a string has the following characteristics as compared to playback using a sound ID. Use string playback according to your needs.

- Because the sound ID file is not included, programs do not need to be recompiled even if sound data is updated.
- It is necessary to load label string data.
- There is execution cost due to converting the string into a sound ID.
- Spelling mistakes in sound specifications cannot be detected during compilation.

Be sure to refer to the sample demo titled `labelString`, which provides an example of playback using a string.

3.10 Types of Sounds

There are three different types of sounds that can be played with the sound library:

- Stream sounds
- Wave sounds
- Sequence sounds

All of these sounds can be played using the `nw::snd::SoundArchivePlayer::StartSound` function. Thus, the programmer does not need to be concerned about which types of sounds were created by the sound designer. Moreover, functions are prepared for all of the basic operations of the sound handle (for example, pausing, stopping, and changing the volume and pitch) so common operations can be used to control playback.

To perform operations that can only be performed on sequence sounds, such as changing the tempo, use a sequence sound handle. Sequence sound handles are treated the same as regular sound handles, but they have additional functions specific to sequence control.

To use a sequence sound handle, pass the sound handle to use as an argument and call the `nw::snd::SequenceSoundHandle` class constructor. Then call functions for specific sequence operations.

Code 3-18 Using Sequence Sound Handles

```
if ( s_SoundArchivePlayer.StartSound( &s_SoundHandle,  
SEQ_MARIOKART ).IsSuccess() )  
{  
    nw::snd::SequenceSoundHandle seqHandle( &s_SoundHandle );  
    seqHandle.SetTempoRatio( 2.0f );  
}
```

If a handle bound to a sound other than a sequence sound is passed to an `nw::snd::SequenceSoundHandle` class constructor, that sequence sound handle will be disabled. Function calls for disabled sequence sound handles are ignored.

4 Memory Management

Memory management plays a very small role in the simple demo. The only time memory management is involved in this demo is when the heap is built during initialization and later, when the data is loaded to the heap.

This chapter provides information about managing memory.

4.1 How to Use the Sound Heap

Since the sound heap class `nw::snd::SoundHeap` is a stack-type heap (LIFO), memory space is reserved in order from the top and released in order from the bottom. Memory from the heap is automatically reserved when sound data is loaded. This process is carried out by the `nw::snd::SoundDataManager::LoadData` function.

Sound data that is no longer needed is deleted by releasing the corresponding region of memory. There are two ways to release the memory region:

- Clear all
- Restore prior state

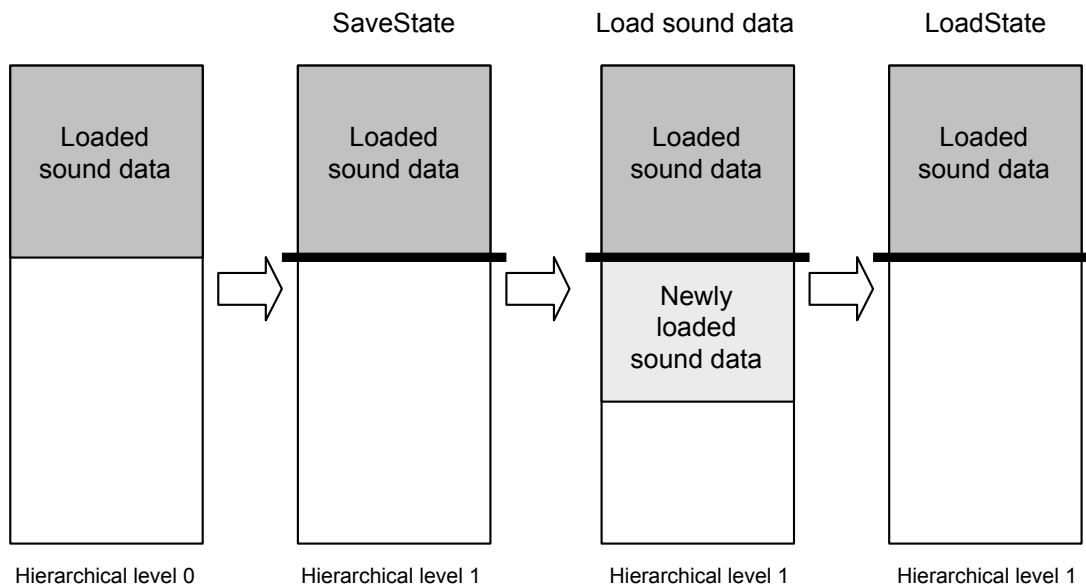
4.1.1 Clear All

This deletes all sound data and restores the initial state. To clear all sound data in the sound heap, call the `nw::snd::SoundHeap::Clear` function.

4.1.2 Restore Prior State

Use this method to delete only the unnecessary data. First, save the current state by calling the `nw::snd::SoundHeap::SaveState` function. As the return value, this function returns the hierarchical level, which indicates the state after saving. You can then use this value to restore the state to the state of the saved heap.

If, after loading a number of sets of sound data, you call the `nw::snd::SoundHeap::LoadState` function using this above-mentioned value for the hierarchical level, you can return to the state that existed immediately after the call to `nw::snd::SoundHeap::SaveState`. This has the effect of deleting all data that was loaded after the call to `nw::snd::SoundHeap::SaveState`.

Figure 4-1 The LoadState Function Can Be Used to Restore a Prior State

At this point, you can continue playing the sound that was being played using the loaded sound data. The `nw::snd::SoundHeap::SaveState` function can be called repeatedly, and each time the hierarchical level increases in value.

4.2 Sound Heap Application

4.2.1 Feature to Automatically Stop Invalid Sound

When using data on the sound heap to playback sound, if the region where that data is stored is released, the playback of that sound is automatically stopped. Therefore, it is unnecessary for the application to confirm whether the sound being played has stopped before a region is released.

However, because the sound is automatically stopped immediately, it may sound unnatural. In these cases, have the application apply a fade-out before releasing the data storage region.

4.2.2 Multiple Sound Heaps

By preparing multiple sound heaps you can save and restore the state of each heap independently. To use multiple sound heaps, all you need to do is create multiple instances of `nw::snd::SoundHeap` and build the heaps using the `nw::snd::SoundHeap::Create` function. You always need to pass a sound heap to the `nw::snd::SoundDataManager::LoadData` function, so this gives you a way to specify the sound heap from which to reserve the memory.

4.2.3 SoundMemoryAllocatable Class

Up to this point, it has been explained that the sound heap must be passed to `nw::snd::SoundDataManager::LoadData`, but strictly speaking it is the

`nw::snd::SoundMemoryAllocatable` class and not the `nw::snd::SoundHeap` class that must be passed. However, because the `nw::snd::SoundHeap` class inherits the `nw::snd::SoundMemoryAllocatable` class, it can be passed as is.

The `nw::snd::SoundMemoryAllocatable` class is an interface class where the `Alloc` function is defined as a purely virtual function. By having the application independently implement the `Alloc` class, memory can be secured from heaps outside of the `nw::snd::SoundHeap` class.

However, when the `nw::snd::SoundHeap` class is not used, be aware that the “Feature to Automatically Stop Invalid Sound” does not work. In this case, if the sound data being played is mistakenly discarded, unexpected sound output may occur.

4.3 Player Heap

4.3.1 The Player Heap

In addition to the sound heap, a player heap is also available.

The player heap temporarily stores data only for sound playback. It is automatically allocated when sound playback begins and automatically deallocated when playback stops. All sound data can be loaded in the player heap.

4.3.2 Using the Player Heap

Because the allocating and deallocating of the player heap occurs automatically in the library, the programmer does not need to manage the player heap. In addition, the sound archive created by the sound designer determines whether the player heap is used.

When sound is played using the player heap, the data load occurs automatically. The data load process occurs asynchronously with the sound data load thread created with the `nw::snd::SoundSystem::Initialize` function and not with a thread called by a sound playback start function (`nw::snd::SoundArchivePlayer::StartSound`, and so on). Accordingly, sound does not start to play immediately after calling the sound playback start function, but sound automatically starts playing after the data load completes. To control the timing when sound starts to play, the `PrepareSound` function can be used in place of the `StartSound` function. For details on the `PrepareSound` function, see Section 3.8 Prepare Sounds (Prepare for Playback).

4.3.3 Memory for the Player Heap

The memory region for the player heap is allocated with the `nw::snd::SoundArchivePlayer::Initialize` function. The amount of memory to allocate to the player heap is based on the values set for Heap Size and Sound Limit in the Player List on SoundMaker.

You can get the values set for Heap Size and Sound Limit with the `nw::snd::SoundArchive::ReadPlayerInfo` function. The memory size for the heap player is

calculated based on these values and they are reflected in value obtained by the `nw::snd::SoundArchivePlayer::GetRequiredMemSize` function.

For player heaps, exactly the maximum number of sound playback times for that player will be allocated.

4.3.4 Cautions for Playback with Different Players

For sound playback, you can pass the `SoundStartable::StartInfo` structure and play the sound using the player that is set for `playerId` in that structure. By doing this, you can play sounds using a player specified by the program and not the player specified by `SoundMaker`. However, you need to exercise caution if you are using the player heap.

The player heap is managed separately for each player, so the existence of a player heap and its size differs by player. Accordingly, if a sound using a player heap is played on a player that either does not have a player heap or has a player heap that is too small, playback will fail.

4.3.5 Data Loaded in the Player Heap

When using the player heap, data loading is performed automatically during playback, but referenced banks and waveform archives are loaded in addition to the sounds. In addition, when the separate load feature for waveform archives is used, the necessary waveform data is loaded separately.

If the total size of the loaded data exceeds the memory size of the player heap, playback will fail. However, caution is required because the loaded data varies according to the sound type and whether separate loading occurs. This section describes what kind of data is loaded with which settings and how to respond when playback fails.

4.3.5.1 Wave Sounds (when separating loading is not used)

When not using the waveform archive separate loading feature, the waveform data necessary for playback is contained in the waveform archive. Thus, the following two types of data are loaded.

- Binary wave sound data (`.bfwsd`)
- Binary waveform archive data (`.bfwar`)

4.3.5.2 Wave Sounds (when separating loading is used)

When using the waveform archive separate loading feature, only the waveform data necessary for playback is loaded separately. Thus, the following three types of data are loaded.

- Binary wave sound data (`.bfwsd`)
- Binary waveform archive data (`.bfwar`) header information
- Binary waveform data referenced by the wave sounds (`.bfwav`)

4.3.5.3 Sequence Sounds (when separating loading is not used)

Sequence data references a maximum of four bank data units. Also, because bank data references each waveform archive data, the following three to a maximum of nine types of data are loaded.

- Binary sequence data (`.bfseq`)
- Maximum of four units of binary bank data (`.bfbnk`)
- Maximum of four units of binary waveform archive data (`.bfwar`)

4.3.5.4 Sequence Sounds (when separating loading is used – Not Recommended)

The maximum number of files that can be loaded to the player heap is nine. If this limit is exceeded, playback may not be correct.

With sequence data, a single bank data unit may reference several waveform data units, and the upper limit may be exceeded when using the waveform archive separate load feature. The following four types of data are loaded, but Nintendo recommends that the separate loading be used within a range that does not exceed the upper limit or that separate loading is not used.

- Binary sequence data (`.bfseq`)
- Maximum of four units of binary bank data (`.bfbnk`)
- Maximum of four units of binary waveform archive data (`.bfwar`) header information
- Binary waveform data referenced by the banks (`.bfwav`)

4.3.5.5 Handling Insufficient Player Heap Memory

When playback fails due to insufficient memory, you can handle this by either increasing the player heap memory size or decreasing the size of the loaded data.

Use the following methods to decrease the size of the loaded data. The first method can be used for wave sounds, while the second method can be used for both wave sounds and sequence sounds.

- The size of the binary wave sound data (`.bfwsd`) increases when the number of wave sounds included in the wave sound set is large. In this case, the size can be decreased by dividing the wave sound set.
- In the same way, the size of the binary waveform archive (`.bfwar`) and its header information increases when the number of waveforms included in the waveform archive is large. In this case, the size can be decreased by dividing the waveform archive.

However, there are cases when more memory than the size of the binary data is required due to alignment or the management region. In such cases, be aware of the following.

- For player heap loading, memory is allocated using 64-byte alignment.
- When setting separate loading, a management region of $(4 * \text{number of waveforms} + 4)$ bytes is required for each waveform archive and a management region of eight bytes is required for each separate download of a waveform.

5 Sound Actors

This chapter describes using sound actors that manage sound for each actor.

5.1 Sound Actors

Some characters in a game may have audible footsteps as they walk or talk as they swing a sword. At this time, the sounds of the footsteps, voice, and sword are made by a single character. Sound actors collectively manage the multiple sounds produced by a single character.

The following are possible when using sound actors.

- When a character disappears from a game, all of the sounds that the character makes can be stopped together.
- All of the parameters, such as volume, for the sounds produced by a character can be changed at once.
- The number of sounds that one character can make simultaneously can be limited.

5.2 The SoundActor Program

This section describes the sound program and the source code.

The sound program is in the file `main.cpp` located in the `%NW4F_ROOT%\Demo\snd\soundActor` directory.

5.2.1 Initializing the Instances

The program creates instances of and initializes the sound actor class (`nw::snd::SoundActor`).

Code 5-1 SoundActor Instance Initialization

```
// SoundActorApp.h
nw::snd::SoundActor s_SoundActor;

// SoundActorApp.cpp
s_SoundActor.Initialize( s_SoundArchivePlayer );
```

A reference to the sound archive player (`nw::snd::SoundArchivePlayer`) is passed as an argument of the `nw::snd::SoundActor` class `Initialize` function. When `nw::snd::SoundActor` is used to play back sound, the result is that the `nw::snd::SoundArchivePlayer` is used for playback.

5.2.2 Sound Playback Using Sound Actors

To play back sound using sound actors, use the `nw::snd::SoundActor` class in place of the `nw::snd::SoundArchivePlayer` class for playback. The `nw::snd::SoundActor` class has the same playback functions as the `nw::snd::SoundArchivePlayer` class.

Code 5-2 Sound Playback Using Sound Actors

```
bool result = m_Actor.StartSound( &s_Soundandle, SE_YOSHI ).IsSuccess();
```

5.2.3 Setting the Maximum Number of Sounds to Be Played Simultaneously

The maximum number of sounds that can be played per actor can be set. The default is an unlimited number of sounds.

Code 5-3 Setting the Maximum Number of Sounds to Be Played Simultaneously

```
s_SoundActor.SetPlayableSoundCount( 0, 2 );
```

The first argument is the actor player number. The actor player is described in detail below.

The second argument is the maximum number of sounds that can be played simultaneously. In this example, up to two sounds can be played simultaneously.

5.2.4 Parameter Updates

The volume and pan values for each actor can be changed.

Code 5-4 Parameter Updates

```
s_SoundActor.SetVolume( s_ActorVolume );  
s_SoundActor.SetPan( s_ActorPan );
```

5.2.5 Deallocating Sound Actors

When a `nw::snd::SoundActor` instance is destroyed, such as when an actor is removed, the sound played by that actor continues to play. To stop the sound, the

`nw::snd::SoundActor::StopAllSound` function must be called explicitly.

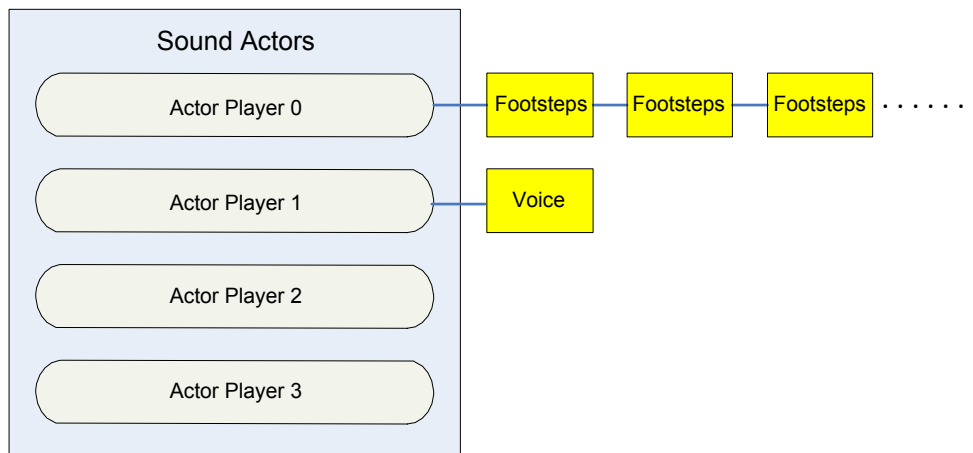
5.3 Actor Players

5.3.1 Maximum Number of Simultaneous Sounds Played and the Actor Player

Although it was explained that the maximum number of sounds that are played simultaneously can be set for each sound actor, more precisely, the maximum can be set for each actor player.

Each sound actor has four actor players. A maximum number of sounds that are played simultaneously can be set for each actor player, from number 0 to number 3.

For example, if there is a character that has footsteps and speaks, there are cases when it is desirable to limit the voice to one and have an unlimited number of footsteps. For this case, set only actor player number 1 to a maximum number of sounds that are played simultaneously to one and set the voice to playback with actor player number 1. Do not set a limit for actor player number 0, and set the footsteps to playback on actor player number 0.

Figure 5-1 Actor Player

When a sound actor instance is created, the maximum number of sounds to be played simultaneously is set to unlimited for actor player number 0, but is limited to one each for actor player numbers 1, 2, and 3.

5.3.2 Setting the Number of the Actor Player to Be Played

Set the number of the actor player to be played in the sound data as a parameter for each sound. The default setting is actor player number 0.

This can be set in the Actor Player column in the SoundMaker program.

5.4 Dynamically Changing the Sound ID

If a sound actor is used, the sound being played can change dynamically according to the actor status.

For example, when footsteps are being played, it may be desirable to change the sound being played according to the type of surface where the actor is walking. Normally, it is enough to change the sound ID passed in the arguments to the StartSound function, but you can also change the ID of the sound for playback by overriding the `nw::snd::SoundActor::SetupSound` function.

5.4.1 The SoundActor::SetupSound Function

The `nw::snd::SoundActor::SetupSound` function is a virtual function. You can inherit from the `nw::snd::SoundActor` class and then override this function.

Code 5-5 The SoundActor::SetupSound Function

```
virtual SoundStartable::StartResult SetupSound(
    SoundHandle* handle,
    u32 soundId,
    const SoundStartable::StartInfo* startInfo,
    void* setupArg
);
```

The `nw::snd::SoundActor::SetupSound` function is called when playing back sounds. The arguments are mostly the same as for the `StartSound` function. The fourth argument `setupArg` takes the parameters needed for sound setup. When calling the `SetupSound` function of the base class, this argument must be passed without being altered.

5.4.2 Changing the Sound ID

Here, we use the `MySoundActor` class, which inherits from the `nw::snd::SoundActor` class, to show how to change the sound of jumping on the ground depending on the type of ground. By way of example, we implement the `MySoundActor::SetupSound` function.

Code 5-6 Dynamically Changing the Sound ID

```
nw::snd::SoundStartable::StartResult MySoundActor::SetupSound(
    SoundHandle* handle,
    u32 soundId,
    const SoundStartable::StartInfo* startInfo,
    void* setupArg
)
{
    if ( soundId == SE_BOUND_BASE )
    {
        switch ( m_FloorType )
        {
            case FLOOR_TYPE_SAND:
                soundId = SE_BOUND_SAND;
                break;
            case FLOOR_TYPE_WATER:
                soundId = SE_BOUND_WATER;
                break;
            case FLOOR_TYPE_WOOD:
                soundId = SE_BOUND_WOOD;
                break;
        }
    }

    return nw::snd::SoundActor::SetupSound(
        handle,
        soundId,
        startInfo,
        setupArg
    );
}
```

6 3D Sound

This section describes using 3D Sound that changes the volume and pan of sound according to 3D spatial coordinates.

6.1 3D Sound Structure

The 3D sound structure is comprised of the following three classes:

- The 3D Sound Manager (`nw::snd::Sound3DManager`)
- The 3D Sound Actor (`nw::snd::Sound3DActor`)
- The 3D Sound Listener (`nw::snd::Sound3DListener`)

6.1.1 The 3D Sound Manager (`nw::snd::Sound3DManager`)

The 3D sound manager calculates and manages 3D sound parameters. Basically, one instance is generated and used.

6.1.2 The 3D Sound Actor (`nw::snd::Sound3DActor`)

The 3D sound actor represents a single sound source. It sets 3D spatial coordinates and manages sound played by a sound source.

When playing back with 3D sounds, sound is played using a 3D sound actor instead of with the sound archive player (`nw::snd::SoundArchivePlayer`).

The `Sound3DActor` class inherits the `SoundActor` class.

6.1.3 The 3D Sound Listener (`nw::snd::Sound3DListener`)

The 3D sound listener represents a microphone or human ear. To set the listener position and orientation, a transformation matrix called the listener matrix is used.

6.2 3D Sound Program

In this section, the sound program is described along with the source code.

The source code is in the file `main.cpp` located in the `%NW4F_ROO%\Demo\snd\sound3d` directory.

6.2.1 Creating Instances

Instances of the three classes described above are created.

Code 6-1 Creation of 3D Sound-Related Instances

```
nw::snd::Sound3DManager    s_Sound3DManager;  
nw::snd::Sound3DActor      s_Sound3DActor;  
nw::snd::Sound3DListener   s_Sound3DListener;
```

Normally, only one instance each of `nw::snd::Sound3DManager` and `nw::snd::Sound3DListener` is created, and a `nw::snd::Sound3DActor` instance must be created for each sound source.

6.2.2 Initializing the 3D Sound Manager

The initialization of `nw::snd::Sound3DManager` is done like this:

Code 6-2 3D Sound Manager Initialization

```
// Initialize 3D sound manager
{
    size_t setupSize = s_Sound3DManager.GetRequiredMemSize( &s_SoundArchive );
    s_pMemoryFor3dManager = allocator.Alloc( setupSize, 4 );
    s_Sound3DManager.Initialize( &s_SoundArchive, s_pMemoryFor3dManager,
    setupSize );
    s_Sound3DManager.SetMaxPriorityReduction( 32 );
    s_Sound3DManager.SetSonicVelocity( 340.0f / 60 );
}
```

See Section 6.3 Doppler Effect for details on the `nw::snd::Sound3DManager::SetSonicVelocity` function.

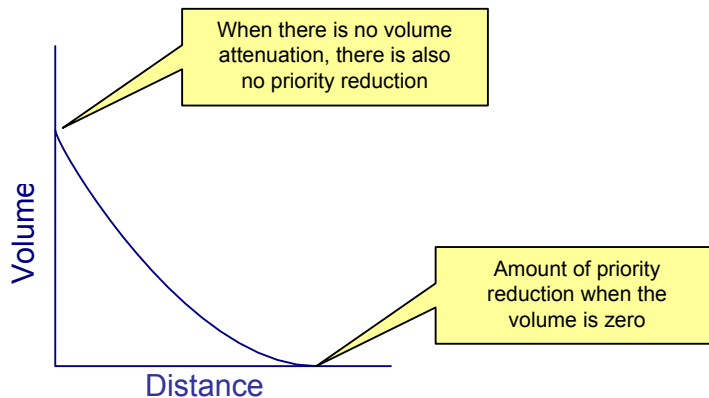
6.2.2.1 Initializing

The `nw::snd::Sound3DManager::Initialize` function initializes the 3D sound manager. The arguments take a pointer to the sound archive being used and the memory region required for initialization. The required memory region size can be obtained using the `nw::snd::Sound3DManager::GetRequiredMemSize` function.

6.2.2.2 Maximum Priority Reduction

The maximum priority reduction is set using the `nw::snd::Sound3DManager::SetMaxPriorityReduction` function.

The player priority of the sound being played as a 3D sound normally decreases in relation to the drop in volume. The maximum priority reduction represents the player priority reduction when the volume drops to 0.

Figure 6-1 Reducing the Player Priority Value

6.2.3 Initializing the Listener

The initialization of `nw::snd::Sound3DListener` is done like this:

Code 6-3 Initializing the Listener

```
// Initialize 3D sound listener
{
    s_Sound3DManager.AddListener( &s_Sound3DListener );

    CalcListenerMatrix( &s_ListenerMtx );
    s_Sound3DListener.SetMatrix( s_ListenerMtx );
    s_Sound3DListener.SetUnitBiquadFilterValue( 0.8f );
    s_Sound3DListener.SetMaxVolumeDistance( 2.0f );
    s_Sound3DListener.SetUnitDistance( 5.0f );
    s_Sound3DListener.SetInteriorSize( 5.0f );
}
```

6.2.3.1 Listener Registration

Register the listener in the 3D sound manager using the `nw::snd::Sound3DManager::AddListener` function.

Multiple listeners may also be registered in a single 3D sound manager. For details, see Section 6.4 Multi-Listeners.

6.2.3.2 The Listener Matrix

Set the listener matrix using the `nw::snd::Sound3DListener::SetMatrix` function. In the sample demo, the listener matrix is determined using the following procedure.

Code 6-4 Calculating the Listener Matrix

```
void CalcListenerMatrix( nw::math::MTX34* mtx )
{
    const nw::math::VEC3 pos( 0.0f, 0.0f, -3.0f ); // Listener position
```



```

const nw::math::VEC3 upVec( 0.0f, 1.0f, 0.0f ); // Listener up vector
const f32 degree = 0.0f; // Listener facing

// Listener directional vector
const nw::math::VEC3 direction(
    -nw::math::SinDeg( degree ), 0.0f, -nw::math::CosDeg( degree )
);
nw::math::VEC3 target = pos + direction;

// Generate listener matrix
nw::math::MTX34LookAt( mtx, &pos, &upVec, &target );
}

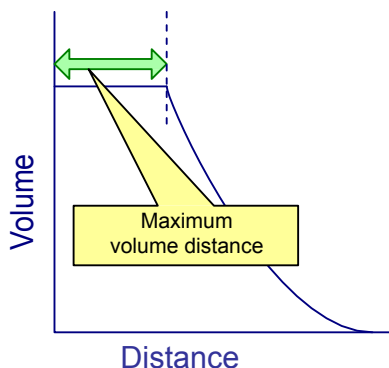
```

6.2.3.3 Maximum Volume Distance

The maximum volume distance is set with the `nw::snd::Sound3DListener::SetMaxVolumeDistance` function.

The maximum volume distance is the distance within which the volume remains at its maximum level, even if the position changes. It is specified as a distance from the listener position.

Figure 6-2 Maximum Volume Distance



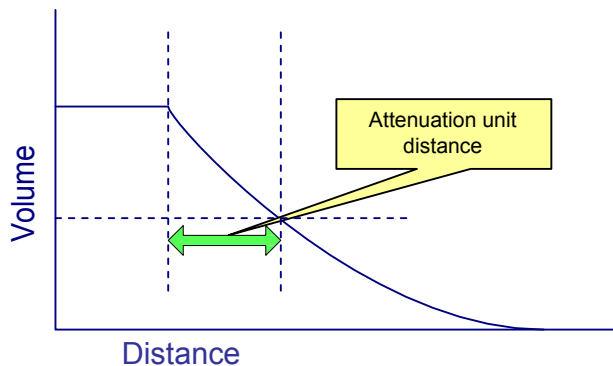
By setting the maximum volume distance, you can prevent the volume from changing dramatically when an actor's position changes near the listener.

6.2.3.4 Attenuation Unit Distance

The attenuation unit distance is set using the `nw::snd::Sound3DListener::SetUnitDistance` function.

The attenuation unit distance indicates the distance at which the volume falls to approximately half its value. Because the maximum volume distance is not included in this distance, the volume falls to half when the actor moves to a distance from the listener that is (maximum volume distance) + (attenuation unit distance).

Although volume falls to half at the attenuation unit distance by default, the volume attenuation rate at the attenuation unit distance can be changed for each separate sound by manipulating the sound data.

Figure 6-3 Attenuation Unit Distance

6.2.3.5 Interior Size

The interior size is set with the `nw::snd::Sound3DListener::SetInteriorSize` function.

The interior size is the size of the area in which the pan and surround pan vary. It is set as a (radial) distance from the listener's position.

If the interior size is large, pan changes become smooth. Conversely, if the interior size is small, pan changes become sudden.

Note: Although sound volume is attenuated as the distance from the listener increases, the amount of attenuation depends on the settings made for the maximum sound volume range (described above).

6.2.4 Initialize the Actor

The initialization of `nw::snd::Sound3DActor` is done like this:

Code 6-5 Initialize the Actor

```
// Initialize 3D sound actor
{
    s_Sound3DActor.Initialize( s_SoundArchivePlayer, s_Sound3DManager );
    s_ActorPos = nw::math::VEC3::Zero();
    s_Sound3DActor.SetPosition( s_ActorPos );
}
```

The arguments to the `nw::snd::Sound3DActor::Initialize` function take references to a sound archive player (`nw::snd::SoundArchivePlayer`) and a 3D sound manager (`nw::snd::Sound3DManager`). Playing back a sound using the `nw::snd::Sound3DActor` function effectively plays back the sound using the `nw::snd::SoundArchivePlayer` object passed in the arguments. Similarly, the 3D sound parameters for the sound being played back are calculated from the `nw::snd::Sound3DManager` object passed in the arguments.

6.2.4.1 Actor Coordinates

The actor coordinates are set with the `nw::snd::Sound3DActor::SetPosition` function.

6.2.5 3D Sound Playback

All 3D sounds are played back using `snd::Sound3DActor` rather than using `nw::snd::SoundArchivePlayer`. The `snd::Sound3DActor` class has a sound playback function similar to `nw::snd::SoundArchivePlayer`.

Code 6-6 3D Sound Playback

```
bool result = s_Sound3DActor.HoldSound( &s_SoundHandle,  
SE_IDLE ).IsSuccess();
```

`Sound3DActor` inherits `SoundActor`, so you can use the same method described in 5.4 Dynamically Changing the Sound ID to do things like change the footstep sound ID to play different kinds of sounds depending on the surface on which the character is walking.

6.2.6 Updating the Actor's Coordinates

Updating the actor's coordinates as needed will also update the 3D parameters of the sound being played.

Code 6-7 Updating Actor Coordinates

```
s_Sound3DActor.SetPosition( s_ActorPos );
```

The listener matrix can be updated using the same method as the actor's coordinates.

6.2.7 Lifespans of Actors and Sounds

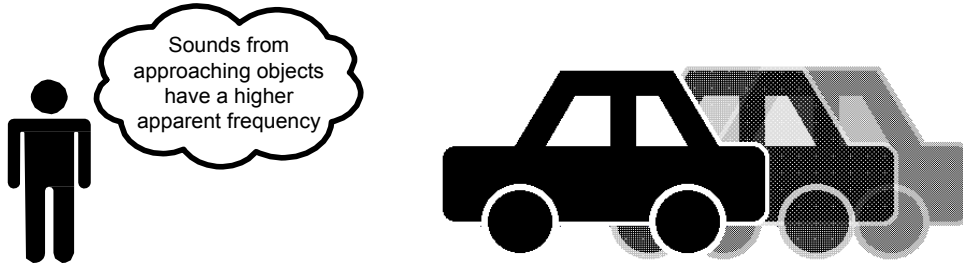
Even if an instance of `nw::snd::Sound3DActor` is destroyed (for example, when an actor disappears), the sound that was played by the actor will continue to play. Also, because position information is stored on the sound side, 3D sound parameters will be calculated correctly so that the sound will play at the position where the actor disappeared, even if the listener matrix is updated.

Accordingly, programmers do not need to worry about lifetimes of `nw::snd::Sound3DActor` instances.

For example, if position data is not updated continuously, the instance of `nw::snd::Sound3DActor` could be destroyed right after playing the sound.

6.3 Doppler Effect

The following describes the method of representing the Doppler effect, where the pitch of a sound depends on the relative velocity of the object producing the sound and the person hearing it.

Figure 6-4 Doppler Effect

The sound3d sample demo is provided as a reference for the Doppler effect.

6.3.1 Doppler Effect Parameters

The following two parameters must be set to apply the Doppler effect.

- Speed of sound
- Doppler factor

6.3.1.1 Speed of Sound

The speed of sound must be set to calculate the Doppler effect.

Code 6-8 Setting the Speed of Sound

```
s_Sound3DManager.SetSonicVelocity( 340.0f / 60 );
```

The default value is 0.0f, meaning the Doppler effect is not applied.

The unit for the value is the speed of sound per frame. The speed of sound is approximately 340 m/sec. So if for the coordinates set by the `Sound3DActor::SetPosition` function 1.0f is equal to 1 meter, then the value should be set to $340.0f / 60$ if the `SoundArchivePlayer::Update` function is being called with coordinates updated at a rate of 60 fps.

6.3.1.2 Doppler Factor

The Doppler factor is a parameter to adjust the amount of Doppler effect to apply.

The Doppler factor is set as a parameter for each sound in the sound data. The default value is 0, but this value can be increased to 127 to apply the Doppler effect. A standard value is 32.

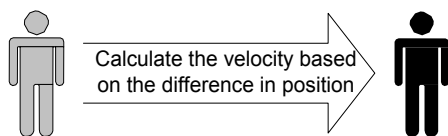
This can also be set using the Doppler Effect column in the SoundMaker program.

6.3.2 Changing the Tone

When the two parameters above are set, the Doppler effect is automatically applied.

Specifically, in either the `Sound3DActor::SetPosition` or `Sound3DListener::SetMatrix` functions, speed is calculated with the difference from the previous call, and then the Doppler effect is applied. It is unnecessary for the speed to be set by the application.

Figure 6-5 Automatic Calculation of Speed



6.3.3 Explicitly Specifying Speed

As mentioned above, speed is automatically calculated. But there may be cases where speed needs to be explicitly set by the application.

For example, if the camera position jumps to another location, a sudden pitch change may occur because the speed is calculated by the distance jumped and processing assumes an extreme speed. In this case, after the `Sound3DListener::SetMatrix` function is called, the speed must be explicitly specified with the `Sound3DListener::SetVelocity` function.

In addition, when generating a `Sound3DActor` that already has speed, the speed must be explicitly specified.

6.3.4 Formula for Pitch Change

The amount that pitch is changed is found with the following formula.

$$pitch = \frac{V_s - V_l \times F}{V_s - V_a \times F}$$

<i>pitch</i>	Ratio of pitch change
<i>V_s</i>	Speed of sound
<i>V_l</i>	Speed of the listener
<i>V_a</i>	Speed of the actor
<i>F</i>	Doppler Factor ÷ 32

6.4 Multi-Listeners

Multiple listeners can be registered in a single 3D sound manager. The operational specifications are different for one listener and for multiple listeners, so to distinguish between the two, a single listener is called a single listener and multiple listeners are called multi-listeners.

Multi-listeners are used when the game screen is divided and displayed in several sections. When the screen is divided, several cameras come into existence, so multiple listeners must also be registered.

6.4.1 Differences Compared to a Single Listener

A multi-listener differs in the following ways from a single listener.

- The volume takes the maximum value of the results calculated for each listener.
- The player priority is the highest priority value of the results calculated for each listener.
- Pan and surround pan are not changed.
- The Doppler effect cannot be applied.

6.5 3D Sound Customization

Up to this point, the standard operations of 3D sound have been described, but many of these can be freely customized. For example, the volume attenuation curve due to distance can be either logarithmic or linear, but can be customized to any curve.

6.5.1 3D Sound Engines (Sound3DEngine)

The calculation processing for 3D sound is performed by a 3D sound engine. One 3D sound engine is allocated to the 3D sound manager. The 3D sound engine implements a default 3D sound calculation process, but a proprietary 3D sound calculation process can be performed by customizing it.

6.5.2 Creating a 3D Sound Engine

To create a proprietary 3D sound engine, create a class that inherits the `Sound3DEngine` class.

Code 6-9 3D Sound Engine Inheritance

```
class MySound3DEngine : public nw::snd::Sound3DEngine
{
    ...
}
```

The created `MySound3DEngine` class instance is registered in `Sound3DManager`.

Code 6-10 3D Sound Engine Registration

```
MySound3DEngine s_MySound3DEngine;

s_Sound3DManager.SetEngine( &s_MySound3DEngine );
```

6.5.3 Implementing 3D Sound Calculation Processing

In order to employ 3D sound calculation processing, the `Sound3DEngine::UpdateAmbientParam` function must be overridden.

Code 6-11 Overriding the UpdateAmbientParam Function

```
class MySound3DEngine : public nw::snd::Sound3DEngine
{
    ...
}
```

```
virtual void UpdateAmbientParam(  
    const nw::snd::Sound3DManager* manager,  
    const nw::snd::Sound3DParam* actorParam,  
    u32 soundId,  
    u32 updateFlag,  
    nw::snd::SoundAmbientParam* param  
);  
}
```

This function is periodically called to update the 3D sound parameters. If this function is called, the content of the `SoundAmbientParam` structure (the last argument) is updated as needed.

Each argument is described in the sections below.

6.5.3.1 Sound3DManager

The registered listener list can be gotten from `Sound3DManager`. 3D sound calculation is based on this listener information.

6.5.3.2 Sound3DParam

The `Sound3DParam` structure stores the `Sound3DActor` coordinates and the 3D sound related parameters set for each sound. When 3D sound is calculated, these parameters must be reflected.

6.5.3.3 Sound ID

The Sound ID can be used as a hint for 3D sound calculation. It can be ignored if it is not needed.

6.5.3.4 Update Flag

This is a bit flag that indicates the parameters that should be updated. Only the parameters with a valid bit are updated.

6.5.3.5 SoundAmbientParam

The `SoundAmbientParam` structure includes the following members.

Code 6-12 The SoundAmbientParam Structure

```
struct SoundAmbientParam  
{  
    f32 volume;  
    f32 pitch;  
    f32 lpf;  
    f32 biquadFilterValue;  
    int biquadFilterType;  
    int priority;  
    u32 userData;  
    ...  
}
```

These structure members are updated based on the data from the four arguments mentioned above. The `fxSend` value specified here is reflected in AUX bus A.

6.5.4 Sound3DCalculator

Because 3D sound calculations are complex, the `Sound3DCalculator` utility class has been prepared to allow for easier customization.

Code 6-13 The Sound3DCalculator Class

```
class Sound3DCalculator
{
public:
    static void CalcVolumeAndPriority(
        const Sound3DManager& manager,
        const Sound3DListener& listener,
        const Sound3DParam& actorParam,
        f32 actorDistance,
        f32* volumePtr,
        int* priorityPtr
    );
    static void CalcPan(
        const Sound3DManager& manager,
        const Sound3DListener& listener,
        const Sound3DParam& actorParam,
        const CalcPanParam& calcPanParam,
        f32* panPtr,
        f32* spanPtr
    );
    static void CalcPitch(
        const Sound3DManager& manager,
        const Sound3DListener& listener,
        const Sound3DParam& actorParam,
        f32* pitchPtr
    );
    ...
};
```

If you use the `Sound3DCalculator` class, you can calculate the 3D sound parameters using the `UpdateAmbientParam` function arguments almost without modification. Reference the implementation code of the `Sound3DCalculator` class for an easy and more detailed customization.

Revision History

Version	Revision Date	Category	Description
1.0.4	2015/07/03	Added	1.2 Multi-Core and Multi-Thread Operations
1.0.3	2013/07/09	Added	4.3.5 Data Loaded in the Player Heap
1.0.2	2013/02/08	Added	Changed the mentions of priority to make it explicit that these refer to player priority.
1.0.1	2012/11/29	Changed	Updated the sample code.
1.0.0	2011/09/07	—	Initial version.

All company and product names in this document are the trademarks or registered trademarks of their respective companies.

© 2011–2013 Nintendo

The contents of this document cannot be duplicated, copied, reprinted, transferred, distributed, or loaned in whole or in part without the prior approval of Nintendo.